1. **Write a parallellized and distributed version of the QuickSort algorithm using MPI.**

```
#include <iostream>
#include <mpi.h>
#include <vector>
#include <algorithm>
#include <time.h>
#include <stdint.h>

using namespace std;

const int tagDataSize = 1;
const int tagInput = 2;
const int tagOutput = 3;

// Tree exploration functions
bool hasChildren(unsigned nrProcs, unsigned /*myId*/, unsigned level)
{
    return ((nrProcs >> level) > 1);
}

unsigned rightChild(unsigned nrProcs, unsigned myId, unsigned level)
{
    return myId + (nrProcs >> (level+1));
}

void parentAndLevel(unsigned nrProcs, unsigned myId, unsigned& parentId,
unsigned& level)
{
    level = 0;
    while(nrProcs > 1) {
        nrProcs >>= 1;
        ++level;
    }
    parentId = myId;
    while((parentId & 1) == 0) {
        parentId >>= 1;
        --level;
    }
    --parentId;
}

// Finds the pivot and splits the src vector into pivot, v1 and v2.
Precondition: src must be non-empty.
void split(vector<int> const& src, int& pivot, vector<int>& v1, vector<int>& v2)
{
    // we try the first, the last and the middle elements, and we choose the
middle value of those 3
    size_t n = src.size();
    size_t mid = n/2;
    size_t pivotIdx;
    if(src[0] < src[mid]) {
        if(src[mid] < src[n-1]) {
            pivotIdx = mid;
        } else if(src[n-1] < src[0]){
            pivotIdx = 0;
        } else {
            pivotIdx = n-1;
        }
    } else {
        if(src[mid] > src[n-1]) {
            pivotIdx = mid;
        } else if(src[n-1] > src[0]){
            pivotIdx = 0;
        } else {
            pivotIdx = n-1;
        }
    }
    v1.clear();
    v2.clear();
    pivot = src[pivotIdx];
    for(size_t i=0 ; i<src.size() ; ++i) {
        if(src[i] < pivot) {
            v1.push_back(src[i]);
        } else if(i != pivotIdx) {
            v2.push_back(src[i]);
        }
    }
}

// Merges back the two sorted vectors and the pivot.
void join(int& pivot, vector<int> const& v1, vector<int> const& v2, vector<int>&
dest)
{
    dest.clear();
    dest.reserve(v1.size()+v2.size()+1);
    std::copy(v1.begin(), v1.end(), std::back_inserter(dest));
    dest.push_back(pivot);
    std::copy(v2.begin(), v2.end(), std::back_inserter(dest));
}

// Performs quicksort on the 'vec'. This is called on process 'myId' at level
'level'; it delegates to sub-processes if needed.
void quicksort(vector<int>& vec, unsigned nrProcs, unsigned myId, unsigned
level)
{
    if(hasChildren(nrProcs, myId, level)) {
        unsigned childId = rightChild(nrProcs, myId, level);
        int dataSize;
        if(vec.empty()) {
            cout << "Process " << myId << " sending empty vector to " << childId
<< "\n";
            // just send a zero data size, so that the child knows it has
nothing to do and ends
            dataSize = 0;
            MPI_Send(&dataSize, 1, MPI_INT, childId, tagDataSize,
MPI_COMM_WORLD);
        } else {
            // We split the work, we give half to the right child, and we keep
half and process it locally
            vector<int> v1;
            vector<int> v2;
            int pivot;
            split(vec, pivot, v1, v2);
            dataSize = v2.size();
            cout << "Process " << myId << " sending " << dataSize << " elements
to " << childId << "\n";
            MPI_Send(&dataSize, 1, MPI_INT, childId, tagDataSize,
MPI_COMM_WORLD);
            if(dataSize > 0) {
                MPI_Send(v2.data(), dataSize, MPI_INT, childId, tagInput,
MPI_COMM_WORLD);
            }
            quicksort(v1, nrProcs, myId, level+1);
            if(dataSize > 0) {
                cout << "Process " << myId << " receiving " << dataSize << "
elements from " << childId << "\n";
                MPI_Status status;
                MPI_Recv(v2.data(), dataSize, MPI_INT, childId, tagOutput,
MPI_COMM_WORLD, &status);
            }
            join(pivot, v1, v2, vec);
        }
    } else {
        // we process locally
        if(!vec.empty()) {
            vector<int> v1;
            vector<int> v2;
            int pivot;
            split(vec, pivot, v1, v2);
            quicksort(v1, nrProcs, myId, level+1);
            quicksort(v2, nrProcs, myId, level+1);
            join(pivot, v1, v2, vec);
        }
    }
}

void readData(vector<int>& v, int argc, char** argv)
{
    unsigned n;
    if(argc != 2 || 1!=sscanf(argv[1], "%u", &n) ){
        fprintf(stderr, "usage: mergesort <n>\n");
        return;
    }

    v.reserve(n);
    for(size_t i=0 ; i<n ; ++i) {
        // v.push_back(rand());
        v.push_back((i*101011) % 123456);
    }
    cout << "generated\n";
}

bool isSorted(vector<int> const& v)
{
    size_t const n = v.size();
    for(size_t i=1 ; i<n ; ++i) {
        if(v[i-1]>v[i]) return false;
    }
    return true;
}

void writeData(vector<int> const& v)
{
    if(isSorted(v)){
        cout << "Ok\n";
    } else {
        cout << "NOT SORTED!!!\n";
    }
}
```

```
int main(int argc, char** argv)
{
    MPI_Init(0, 0);
    int myId;
    int nrProcs;
    MPI_Comm_size(MPI_COMM_WORLD, &nrProcs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myId);

    vector<int> vec;
    unsigned level;
    unsigned parent;
    int dataSize;
    if(myId == 0) {
        readData(vec, argc, argv);
        level = 0;
    } else {
        parentAndLevel(nrProcs, myId, parent, level);
        cout << "Process " << myId << " of " << nrProcs << ", at level " <<
level <<", receiving initial data from " << parent << "\n";
        MPI_Status status;
        MPI_Recv(&dataSize, 1, MPI_INT, parent, tagDataSize, MPI_COMM_WORLD,
&status);
        if(dataSize > 0) {
            vec.resize(dataSize);
            MPI_Recv(vec.data(), dataSize, MPI_INT, parent, tagInput,
MPI_COMM_WORLD, &status);
        }
    }
    quicksort(vec, nrProcs, myId, level);
    if(myId == 0) {
        writeData(vec);
    } else if(!vec.empty()){
        cout << "Process " << myId << " sending final data to " << parent <<
"\n";
        MPI_Send(vec.data(), dataSize, MPI_INT, parent, tagOutput,
MPI_COMM_WORLD);
    } else {
        cout << "Process " << myId << " had nothing to do and is about to
end\n";
    }
    MPI_Finalize();
}
```

2. Consider the following code:

```
struct Account {
    unsigned id;
    unsigned balance;
    mutex mtx;
};

bool transfer(Account& from, Account& to, unsigned amount) {
    unique_lock lck1(from.mtx);
    unique_lock lck2(to.mtx);
    if(from.balance < amount) return false;
    from.balance -= amount;
    to.balance += amount;
}
```

Concurrency issue

There is a potential deadlock in this situation if there are two transfers happending at the same time like this:

```
thread1:
  transfer(account_a, account_b, x)
thread2:
  transfer(account_b, account_a, y)
```

If the two threads can acquire the first lock thei need (the first unique_lock from the method), a deadlock will happen.

Solution

Assuming the ids are unique (obviously), lock the accounts in order.

Code

```
bool transfer(Account& from, Account& to, unsigned amount) {
    if(from.id < to.id) {
      unique_lock lck1(from.mtx);
      unique_lock lck2(to.mtx);
    } else if (from.id > to.id) {
      unique_lock lck1(to.mtx);
      unique_lock lck2(from.mtx);
    } else {
      // tranfer from the same account
      if(from.balance < amount) return false;
      return true;
    }
    if(from.balance < amount) return false;
    from.balance -= amount;
    to.balance += amount;
    return true;
}
```

3. **Give an algorithm for solving the following variant of consensus problem:**

- Each process i has an input value vi;
- All processes must decide some output value; the output value must be the same for all processes and, if all inputs are equal, then the common output must be equal to that input;
- We have synchronous rounds;
- The processes are all correct;
- Messages may be lost; however, if a party sends infinitely many messages, then infinitely many will be delivered;
- Each process must decide after a finite amount of time; however, it can continue the algorithm forever.

Solution

Since no process is faulty, it is enough to make the first process send its value to all other processes and have all processes agree on that value. There is no risk that the first process sends distinct values to distinct processes; however, we must make sure that the value sent by the first process arrives at all other processes.

So, the solution is the following:

process 1 decides y1=x1; Then, forever, it waits for requests from other processes and answer each request by sending back x1 to the requester. each of the other processes repeatedly sends requests to process 1 until it receives an answer. At that moment, it decides the values read from that answer.

1. **Write a parallel (distributed or local, at your choice) program for solving the k-coloring problem**

That is, you are given a number k, and n objects and some pairs among them that have distinct colors. Find a solution to color them with at most n colors in total, if one exists. Assume you have a function that gets a vector with n integers representing the assignment of colors to objects and checks if the constraits are obeyed or not.

Solution

We will map every possible coloring to a number in base `k`, so the numnber `0` represents the colouring `000...000`. We have number in the interval [0, k^n - 1].

Now, each thread `i` will check all the numbers x such that `x % t == i`.

Code

```
bool check(vector <int> colour);

vector <int> to_color(int sol, int n, int k) {
  vector <int> v;
  for(int i = 0; i < n; ++ i) {
    v.push_back(sol % k);
    sol /= k;
  }
  return v;
}

vector <int> solve(int n, int k, int T) {
  int maxi = 1;
  for(int i = 0; i < n; ++ i) {
    maxi *= k;
  }
  vector <int> sol;
  vector <thread> threads;
  for(int t = 0; i < T; ++ t) {
    threads.push_back([&sol, i, n, k](){
      for(int j = i; j < maxi && sol.size() == 0; j += T) {
        vector <int> col = to_color(sol);
        if(check(col, n, k) {
          sol = col;
        }
      }
    });
  }
  for(int i = 0; i < T; ++ i) {
    threads[i].join();
  }
  return sol;
}
```

1. **Consider the following code for inserting a new value into a linked list at a given position**

We assume that insertions can be called concurrently, but not for the same position. Find and fix the concurrency issue. Also, describe a function for parsing the linked list.

```
struct Node {
    unsigned payload;
    Node* next;
    Node* prev;
    mutex mtx;
};

void insertAfter(Node* before, unsigned value) {
    Node* node = new Node;
    node->payload = value;
    Node* after = before->next;
    before->mtx.lock();
    before->next = node;
    before->mtx.unlock();
    after->mtx.lock();
    after->prev = node;
    after->mtx.unlock();
    node->prev = before;
    node->next = after;
}
```

Concurrentcy issue

Here is an example that illustrates a concurrency issue:

```
// the list A -> B
thread1:
  insertAfter(nodeA, x);
thread2:
  insertAfter(nodeA->next, y);
```

It could happend that after these possible solutions:

We would accept the following two cases:

```
A -> x -> y -> B
A -> x -> B -> y
```

However, this could happen this:

`nodeA->x->B`

Solution

Lock the two nodes simulateously

Code

```
void insertAfter(Node* before, unsigned value) {
    Node* node = new Node;
    node->payload = value;
    Node* after = before->next;
    before->mtx.lock();
    after->mtx.lock();
    before->next = node;
    after->prev = node;
    before->mtx.unlock();
    after->mtx.unlock();
    node->prev = before;
    node->next = after;
}
```

3. We have n servers that can communicate with each other

There are events producing on each of them; each event has an associated information. Each server must write a history of all events (server and event info) produced on all servers, and, furthermore, the recorded history must be the same for all servers. Write a distributed algorithm that accomplishes that. Consider the case n = 2 for starting.

Solution

Let's consider the case when `n = 2`. Let's name the processes `A` and `B`.

Every process will keep a `Lamport clock` and pass that on each message.

Suppose an event occurs in the process `A`. Then, process `A` will increate it's timestamp `t` by one, and send that event alongside `t` at the process `B` - this is called `PREPARE` event. Process `B` computes the maximum between its internal clock and the timestamp of the message and adds one. It send back an `OK` with that computed timestamp. Process `A` receives the `OK` and sends back a `COMMIT` message. They both agreed on this value.

The generalization to `n > 2` follows easily.

Each process having an occuring event will:

- broadcast `PREPARE` with the timestamp
- waits for `OKs` from all the other processes
- computes the maximum amongs the `OKs` timestamps
- broadcast `COMMIT` to each process

There is only one little problem. A process may have previously gave an `OK` but did not get any `COMMIT` yet for that event. So, in case another `COMMIT` appears, it can't write that to a file because the `COMMIT` from the previously sent `OK` may be either before, or after the current `COMMIT`. That's why, each process will maintain a list of given `OKs` as well as a list of `COMMITs` to be flushed to disk.

Note. Every tie can be solved by chosing an initial *arbitrary* ordering of the processes. Such as the `PID` or `IP` if they live on different hosts.